Fairly Fast Digitizer Support

Internal ptr design

Fri, Oct 11, 2002

An "internal ptr" is the "object code" that results from compiling a data request. For each ident and listype in a data request, the internal ptr is the structure that is passed to a "read-type routine" that produces the reply data. Initializing a data request is analogous to compiling a source program. A request block is allocated, and its fields are initialized, including an array of internal ptrs that is generated by calling a ptr-type routine. The total number of internal ptrs is the product of the number of listypes and the number of idents in the request, consisting of one array of internal ptrs for each listype represented in the request.

To generate reply data for an active request, the updating code loops over the listypes of the request and, for each listype, it calls the read-type routine that takes as its arguments the array of internal ptrs, the number of such ptrs to process, the number of bytes of data to generate, and a pointer to the answer area into which that data is placed. The point is that the read-type routine does not have to interpret the original ident array including in the request; instead, it scans the internal ptrs array, which is expected to be require less processing. In this way, updating of repetitive reply data is optimized.

As one of the simplest examples, consider a data request that specifies the listype that asks for 2-byte analog readings for an array of channel number idents. The ptr-type routine for this listype constructs a pointer to the reading field of the ADATA table entry for each specified channel. These pointers are the internal ptrs for this case. The job of the read-type routine is to simply read 2 bytes from the address given by the internal ptr; it is obvious that this is an efficient loop. (The alternative would have been to examine each channel number ident, check for valid range, compute the address of the reading field, then finally copy out 2 bytes from that address.) Each read-type routine, which is called for each listype in a request, is constructed as a loop, which optimizes the performance of processing reply data in response to an array of idents.

Until recently, the size of an internal ptr was always 32 bits, typically allowing for a memory address that points to the source of the data sought, as in the example above. A new feature has been added to the underlying data request support that allows a ptr-type to use more than one 32-bit longword to hold its internal ptr structure. Note that the exact layout of an internal ptr only has to be known by the ptr-type routine and its corresponding read-type routine; routines associated with other listypes are not involved. Only a ptr-type routine produces internal ptrs, and only a read-type routine interprets them, resulting in the production of reply data.

The motivation for the above addition to the system code is for support of fairly fast digitizers. The traditional example of such a digitizer has been the IRM 1 KHz digitizer. Now we need to also support a 10 KHz digitizer that was designed along a similar paradigm. The digitizer hardware writes digitized data into a circular buffer without any need for software intervention. Each digitizer supports 64 channels of analog data. The differences are the size of the circular buffer and the digitize rate. For the 1 KHz case, 64 channels are digitized and stored into 128 bytes of memory every millisecond. The size of the buffer is 64KB, which allows for 512 sets of such data to be written, after which the hardware automatically "wraps" to the beginning. For the 10 KHz case, 64 channels are digitized and stored into 128 bytes of memory every 100 μs. The size of the buffer is 2MB, which allows for 16384 sets of such data before it "wraps."

In order to fit into the generic data request scheme that is exemplified by the Classic protocol specification, the access to such digitizer data was designed to interpret the number of bytes requested per channel number ident and the reply period of the request as an implicit specification for how the digitizer data must be sampled to produce the reply data. Suppose one wants to access 1 KHz data sampled at 300 Hz, say, as any selected sampling rate can be supported. Clearly, 300 Hz does not evenly divide 1000 Hz, but the software support samples the 1KHz data every 3rd or 4th sample to fill the user's reply buffer. Let us say that the reply rate is 15 Hz. There is an 8-byte header in the reply data structure immediately followed by pairs of data and time words, where the times are in units of 10 μs. To request 300 Hz data to be delivered at 15 Hz, for example, specify a buffer size sufficient to hold the header and 2 words for each of the (300/15 = 20) data points, a total of (8 + 20*4 = 88) bytes. For each 15 Hz update, the read-type routine samples the circular buffer from the point where it last left off to the present, and it remembers the address of the present point in the circular buffer to be used as a basis for the next reply data update. (Note that the reply period should not be longer than the "wrap" time of the circular buffer, which is about 0.5 seconds for the 1 KHz case or 1.6 seconds for the 10 Kz case.) Note that the internal ptr in this case is not fixed; it is altered by the read-type routine every time it is called.

What information should be made available to the read-type routine via an internal ptr structure for it to generate the reply data? Here is a suitable list:

- type of digitizer interface
- ptr to the digitizer registers
- ptr to the circular buffer
- ptr to the array of time-stamps for each digitizer time "slot," or data set
- digitizer rate, or period
- size of the circular buffer, or the number of sets in the buffer
- number of channels (fixed at 64?)
- ptr to where the reply data last left off
- clock event# to be used as reference for time stamping
- flag to indicate event group rather than individual event

Whew? There is a lot to retain in order to handle multiple cases of this type. This is why the internal ptr structure needed to be expanded for this case. What is a suitable design for this structure?

Field	*Size*	*Meaning*
`extFlag`	1	Ext answ flag, event group flag
`event`	1	Clock event number
`nChans`	2	Number of channels in one set
`digType`	1	Digitizer type#
`digPeriod`	3	Digitizer period in μs (same units as time-stamps)
`bufSize`	4	Size of circular buffer
`dataPtr`	4	ptr to previous data word
`timePtr`	4	ptr to time-stamp array
`cBufPtr`	4	ptr to base of circular buffer
`regsPtr`	4	ptr to digitizer registers

The above specification is 7 longwords in size! But it should permit writing a generic read-type routine. The corresponding ptr-type routine, of course, has to initialize this structure. Some of the information it needs can be found in the `CINFO` table entry structure, namely the digitizer type, the ptr to the registers, the ptr to the circular buffer, and the ptr to the time-stamp array. Fields that depend upon the digitizer type# are the size of the circular buffer, the number of channels in a set, and the digitizer period. The remaining fields are derived from the original ident.

Note that a fictitious digitizer type might specify that the number of channels in a set is 1, which would imply that one has a circular buffer of digitized data sampled at some rate. A `CINFO` entry could be designed to contain the needed parameters.

Is it possible to ask what digitizer support exists for a given channel? This would be analogous to the `FTPMAN` timing query that a client first makes to find out what capabilities exists for a given channel. One can assume that everything is accessible at 15 Hz. But some channels are accessible via other hardware, according to what is found in the `CINFO` entries.

The time stamps used with each data word as produced by the generic read-type routine alluded to in this note are 16 bits wide and are in units of 10 μs. This means that the maximum time they can express is 655 ms. But the reply data structure also includes a header that includes a 4-byte base time-stamp, and each 16-bit time stamp is added to this base time to produce the actual full time-stamp that represents the delay after a selected clock event.